Synchronous programming of device drivers for global resource control in embedded operating systems

Nicolas Berthier, Florence Maraninchi, Laurent Mounier

A

Synchronous Programming of Device Drivers for Global Resource Control in Embedded Operating Systems

NICOLAS BERTHIER, UJF/Verimag
FLORENCE MARANINCHI, Grenoble INP/Verimag
LAURENT MOUNIER, UJF/Verimag

In embedded systems, controlling a shared resource like a bus, or improving a property like power consumption, may be hard to achieve when programming device drivers individually. In this paper, we propose a *global resource control* approach, based on a centralized view of the devices' states. The solution we propose operates on the hardware/software interface. It involves a simple adaptation of the application level, to communicate with the hardware via a *control layer*. The control layer itself is built from a set of simple automata: the device drivers, whose states correspond to functional or power consumption modes, and a controller to enforce global properties. All these automata are programmed using a synchronous language, and compiled into a single piece of C code. We take as example the node of a sensor network. We explain the approach in details, demonstrate its use and benefits with an event-driven or multithreading operating system, and draw guidelines for its use in other contexts.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.1 [**Programming Techniques**]; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.11 [**Software Engineering**]: Software Architectures; D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Algorithms

Additional Key Words and Phrases: Automated control, power-aware implementation, synchronous paradigm, wireless sensor networks

ACM Reference Format:
ACM Trans. Embedd. Comput. Syst. V, N, Article A (March 2013), 25 pages.
DOI = 10.1145/2435227.2435235 http://doi.acm.org/10.1145/2435227.2435235

1. INTRODUCTION

1.1. Resource Control in Embedded Systems

In embedded systems, controlling a shared resource, or improving a global property, may be crucial in some application domains. Consider power consumption in the node of a wireless sensor network (WSN). Optimizing power consumption has a direct effect on the lifetime of the system because the battery cannot be recharged. Choosing devices that can be put in some low consuming mode when nothing happens may offer significant gains in sensor networks, where traffic is quite low. This is the case for radio controllers or micro-controller units (MCU). Once the hardware devices have been selected, programming a node in such

This work has been partially supported by the French ANR project ARESA2 (ANR-09-VERS-017).
Verimag is an academic research laboratory affiliated with: the University Joseph Fourier Grenoble (UJF), the National Center for Scientific Research (CNRS) and Grenoble Polytechnic Institute (Grenoble INP).

DOI 10.1145/2435227.2435235 http://doi.acm.org/10.1145/2435227.2435235

a way that the low-consumption modes of the various devices be well exploited is not easy. The software has a huge impact on the consumption states of the various devices (e.g., the driver of the radio puts it in low consumption mode), but this low-level software is usually designed in a local, per device, way. Information on the consumption states of the various devices is then scattered among several pieces of code (device drivers, protocols, application, or Operating System — OS), and the decisions are necessarily taken in a local, decentralized manner.

1.2. Need for Global Control

Consider a simple sensor network application involving two concurrent tasks. The first one periodically senses the environment using a sensor connected to an Analog-to-Digital converter, and sometimes stores this information in a flash memory. When the collected data satisfy a given property, an alarm is sent to a special node of the network. The second task manages the network by listening to the radio channel, using a radio transceiver device. This part of the software is responsible for routing the packets received to the desired nodes, and sending new alarms upon request from the first task. Sensor network hardware platforms usually provide several devices connected to the MCU using a limited number of buses. Therefore, the two almost independent tasks are in practice constrained by *shared resources* like buses. On the other hand, we may need to reduce the *instantaneous* power consumption, for instance by avoiding situations in which the radio and the flash memory devices are simultaneously in an energy-greedy operating mode. If the control of the radio (resp. the memory) is implemented in the driver of the radio (resp. the memory), the decisions on the total power consumption cannot be done, because there is no place in the software where the global information is available.

In the example application, we may consider two *global properties*: (i) mutual exclusion of the accesses to shared resources like buses; (ii) reduction of instantaneous power consumption. With *global control*, the idea is that all information on the functional and non-functional (e.g. power) states of the devices should be gathered and exploited by a global policy implemented as a *centralized controller*.

1.3. Problem Formulation and Proposal

The problem we consider is the following: given a hardware architecture made of several devices whose functional and non-functional states are known, plus some existing application software (e.g., the protocol stack in the case of sensor networks nodes), how to replace the low-level software (typically the set of drivers) by a *control layer* that implements a global resource management policy? One should still be able to implement the application software as a set of concurrent threads on top of a scheduler, or with event-driven programming. Also, only small changes in existing software should be required, if any.

1.3.1. Synchronous Programming and Controllers. To build the control layer we use *synchronous programming*, which has been studied a lot in the embedded system community, especially for hard real-time safety-critical systems. The family of *synchronous languages* [Benveniste et al. 2003] offers complete solutions, from pure static scheduling to some operating system support [Caspi et al. 2008]. The control layer is designed as a parallel program in a synchronous language, and then statically scheduled by the compiler to produce a piece of sequential C code.

The design of the control layer is inspired by *controller synthesis* techniques [Ramadge and Wonham 1989]. The approach is similar to several proposals that have been made in the family of synchronous languages and tools (see, for instance, [Chandra et al. 2003], [Girault and Rutten 2009] or [Altisen et al. 2003]).

To summarize, the approach is as follows: (i) each device driver is described as a Mealy automaton M_i, whose transitions are labeled by Boolean formulas made of inputs from

both the hardware and the software, and by outputs representing low-level C code; (ii) we specify some global properties, like: "*P: the devices A and B should not be in their highest consuming modes at the same time*"; (iii) the automata of the drivers are made *controllable*, meaning that the absence of an additional input may prevent the automaton from changing states: this yields the family of automata M_i''s; (iv) we build an automaton C, to control the M_i''s in such a way that global properties like P are ensured; (v) C and the M_i''s are programmed in some synchronous language; (vi) the control layer is obtained by *compiling* the parallel composition of C and the M_i''s into a single piece of sequential C code.

When there is a large number of devices to be controlled, or when the global properties are complex, designing the automaton C may become a hard task. This is typically when a tool implementing controller synthesis should be used. We do not use a controller synthesis tool yet, for efficiency reasons, but there exist promising tools. See comments on the use of such tools in Section 6.3.

1.3.2. Programming Model and Adaptation of Existing Software. Implementing a global resource management policy (e.g., "*never have two devices in their highest consuming modes at the same time*") has some intrinsic consequences on programming models: some commands *have* to be refused, if they yield a global state that violates the resource management policy. This happens whatever the solution chosen: (i) using parallel programming and blocking primitives (see Section 7 for proposals of this kind); or (ii) using our proposed approach.

Our approach enables a systematic identification and handling of refused commands: the control layer refuses to execute a command if it results in a global state that violates one of the global properties P; the control layer can implement two options: either the request is canceled, or it is delayed until it becomes acceptable.

Moreover, when the handling of refused commands is application-specific, and cannot be done with a systematic implementation, the control layer can transmit a complete and precise description of the situation (current global state, and state that would be reached by executing the command) to the upper layer.

The whole approach allows the easy reuse of previously written software and operating systems. Porting existing code to our new hardware/software architecture only requires the modification of the software that drives the physical devices (e.g., radio transceiver, flash memory), or that manages a bus: part of the code of device drivers has to be replaced by requests to the control layer. Higher level parts of the original operating system, like a network stack or a file system, are unchanged.

1.4. Contributions and Structure of the Paper

This paper makes four contributions to global resource control in embedded systems: (i) a software architecture based on a *control layer*, between the hardware and the high level software; (ii) a method for obtaining the control layer automatically from a formal description of the device drivers, plus a description of the global properties that should be ensured; (iii) a working implementation of these two ideas, which can be used with an either purely event-driven, or multithreading operating system, seating on top of the control layer; (iv) guidelines for the adaptation of existing software on top of the control layer.

The remainder of the paper is structured as follows: in Section 2, we briefly present the technical background for the definition of the control layer; Section 3 gives the hardware platform example; Section 4 gives the principles of our approach; Section 5 is a case study; Section 6 provides an evaluation of the whole approach, and suggests extensions; Sections 7 and 8 review related work and conclude.

2. BACKGROUND ON SYNCHRONOUS LANGUAGES

The essential points of synchronous languages semantics [Benveniste et al. 2003] can be explained with synchronous products of Boolean Mealy Automata (BMAs), as described

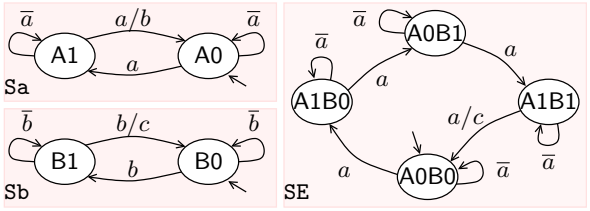

Fig. 1. Two Boolean Mealy Automata `Sa` and `Sb` synchronized via a signal b, and the result of their composition `SE`.

```
node SE (a: bool) returns (c: bool);
var inA0, inA1, mA, inB0, inB1, mB, b: bool;          -- Internal variables.
let
  inA1 = not mA;     inA0 = mA;     b = a and inA1;     -- Encoding of Sa.
  mA = true -> pre (not a and inA0 or a and inA1);     -- (cont'd)
  inB1 = not mB;     inB0 = mB;     c = b and inB1;     -- Encoding of Sb.
  mB = true -> pre (not b and inB0 or b and inB1);     -- (cont'd)
tel;
```

Fig. 2. The two Boolean Mealy automata `Sa` and `Sb` of Figure 1 encoded in Lustre, in a single node that behaves as `SE`. `b` is the only variable shared among the two corresponding sets of equations. `mA` and `mB` are the persistent state variables, the remaining consists of formula aliases for readability, and definition of outputs; e.g., `mA` holds iff `Sa` is in state `A0` (it is initially `true`, being the initial state of `Sa`).

```
/* The reactive kernel produced by the compiler:                              */
int M1, M2, M3;                          /* State variables.                  */
void init () { M3 = 1; }                 /* Initialization.                   */
void run_step (int a)                    /* Step function, calling output procedures. */
{ int L1, L2, L3, L4, L5, L6, L7;
  L3 = M3 | M1; L2 = ~L3; L6 = M3 | M2; L5 = ~L6 & a; L1 = L2 & L5; main_O_c(L1);
  L4 = L3 & ~L5; L7 = L6 & ~a; M1 = L4 | L1; M2 = L7 | L5; M3 = 0;              }
/* To be added to get a main program:                                         */
void main_O_c (int x) { printf ("%d\n", x); } /* Output procedure.            */
int main () {
  init ();                               /* Initialization.                   */
  while (1)                              /* Infinite loop.                    */
  { int a; printf ("Give a (0/1): "); scanf ("%d", &a); /* Get inputs.        */
    run_step (a); } }                    /* Compute next state and produce outputs. */
```

Fig. 3. Reactive kernel for the example of Figure 1, and example main program.

in [Maraninchi and Rémond 2001]. In such automata *inputs* and *outputs* are distinguished, and the communication is based on the asymmetric *synchronous broadcast* mechanism.

Figure 1 is an example. Automaton `Sa` (resp. `Sb`) reads a (resp. b), and emits a b (resp. c) every two a's (resp. b's). `SE` is the result of their composition. It is a automaton that reads a and emits a c every four a's. Note that emitting b in `Sa`, and reacting to b in `Sb`, are combined into a single transition, making communication instantaneous.

In all synchronous languages, a program is made of several components that can be viewed as separate BMAs. Several constructs allow to combine them, in parallel or hierarchically. The various automata communicate via the synchronous broadcast, by sending and receiving signals. From a parallel program, the compilers produce a piece of code called the *reactive kernel*. This kernel has to be wrapped in some loop code, which calls the kernel repeatedly, to make it execute one transition at a time. The C code of the reactive kernel is sequential: the parallelism present in the original program has been *compiled*, i.e., statically scheduled. Note also that the size of the resulting code is linear in the size of the original BMAs, not in the size of their product.

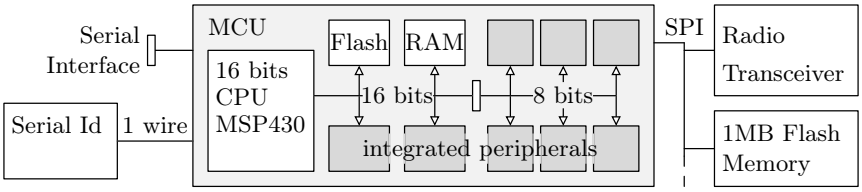

Fig. 4. The Wsn430 hardware platform (taken from [Fraboulet et al. 2007]).

```
1 main ()      /* This function never returns to preserve an execution context. */
2   ...  /* Initialization of the hardware devices and the whole application.   */
3   while (true) enter_lpm ();   /* Enter low-power mode and enable interrupts. */
4   /* Interrupts will awake the MCU, and their handlers will execute on this
5      stack. When they terminate, the low-power mode is immediately reentered. */
```

Fig. 5. Typical structure of the main() function, when programming on bare hardware. Notice interrupts are disabled when this function is called, so the whole initialization process (cf. line 2) is usually uninterrupted.

For the example given above, we can encode the two automata in LUSTRE [Caspi et al. 1987], as illustrated in the listing of Figure 2. We can then use a compiler from LUSTRE to C code. The code produced looks like the one on Figure 3. The first part is the "reactive kernel" produced by the compiler: the run_step and init functions, plus the declaration of the state variables. The second part shows how to use such a kernel, by providing an output function, and calling the kernel in an infinite loop that provides it with inputs. The execution of this code outputs "1" every 4 occurrences of value 1 for the input a.

3. EXAMPLE PLATFORM AND USUAL PROGRAMMING PRACTICES

Figure 4 is a block diagram describing the Wsn430 hardware platform for wireless sensor networks. It is composed of an MSP430 MCU[1] including several integrated peripherals: timers, Analog-to-Digital Converters (ADCs), Universal Synchronous/Asynchronous Receiver/Transmitters (USART) supporting several protocols such as the Serial Peripheral Interface (SPI), etc. This micro-controller has 6 operating modes, among which: the *Active* mode, in which everything is active (this is the mode with the highest consumption); the *LPM4* mode (the one with the lowest consumption), in which there is no RAM retention, the real-time clock is disabled, and the only way to wake up is by an external interrupt; the *LPM3* mode (having an intermediate consumption) in which there is only one peripheral clock available. This MCU can be woken up by any interrupt, including those emitted by its integrated peripherals if they are still active (e.g., ADCs or timers), and external ones.

The Wsn430 platform features a CC1100 radio transceiver[2], a DS2411 serial identifier hardware device[3], a flash memory module, and various sensors. A network simulator, along with a cycle-accurate emulator, can be used in order to test and debug applications and full systems from their target binary code [Fraboulet et al. 2007]. The flash memory module, the radio chip, and a sensor, share the same serial bus. Simultaneous accesses must be avoided.

Programming WSN nodes can be done adopting one of the following practices:

(i) *Programming on the bare hardware* consists in designing the whole software without any system support. Typically, a basic C library is used, with no parallel programming abstraction facility: a single execution context (stack) is provided, in which the main() function starts executing. Figure 5 exemplifies the typical structure of such a main() function. Architecture-dependent code may additionally be used, allowing accesses to low-level features such as MCU operating mode selection (e.g., enter_lpm() on line 3 in Figure 5),

[1] http://focus.ti.com/lit/ug/slau049f/slau049f.pdf; [2] http://focus.ti.com/lit/ds/symlink/cc1100.pdf; [3] http://www.maxim-ic.com/datasheet/index.mvp/id/3711

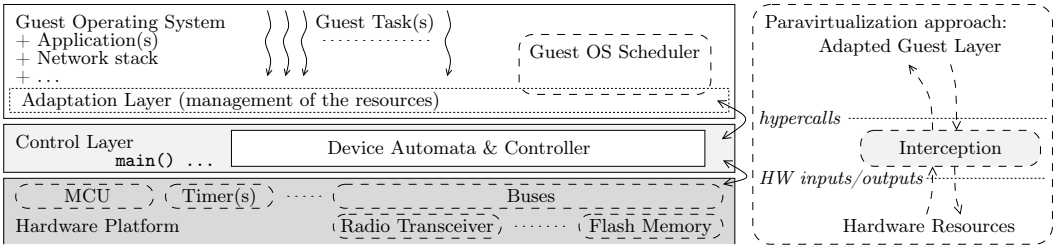

Fig. 6. Global description of the approach.

interrupt management or peripheral device register handling. In this case, all resource management is under application control.

(ii) *Using operating system support* extends the previous approach with the inclusion of *system-level services*. Threading and associated communication mechanisms can be provided by existing OSes, along with advanced services such as networking or file system management. Besides, solutions involving an OS come with their set of available device drivers, and supporting a new hardware platform often requires writing complete new pieces of low-level code. At last, they generally exploit generic mechanisms like locks or monitors for resource and energy management, when they exist; ad hoc solutions are used otherwise, which often require deep and non-trivial modifications of the OS implementation. Widespread operating systems used in WSNs include RETOS [Cha et al. 2007], TinyOS [Hill et al. 2000]. Dunkels et al. [2004] propose Contiki, which also belongs to these systems. It features an event-driven execution model, and concurrent tasks are implemented as stackless cooperating threads called *protothreads* [Dunkels et al. 2006] in order to reduce context switching and stack management costs. Contiki is implemented in C exclusively, so it is a suitable candidate to estimate the impact of the control layer on purely event-driven programs.

In cases (i) and (ii), the usual programming practice is event-based, meaning that separate portions of code communicate with each other by posting *events* in queues, which trigger executions of *handlers* attached to them. Interrupt requests are seen as events that can occur even when the MCU is in a low-power mode. Note also that interrupt handlers can force the MCU to stay in active mode, or to go back in low-power mode, when they return. In this way, the software is written so that the MCU is in a low-power mode most of the time, i.e., when nothing useful is to be computed.

Fault Handling. A WSN node is a quite constrained embedded system, with no hardware redundancy. If a faulty device is able to return an error code, then the fault can be handled by some upper layer. But if a device fails silently, the entire node may fail. In WSN applications, fault-tolerance is considered at the network level, exploiting the redundancy of nodes.

4. PRINCIPLES OF THE SOLUTION

The solution we propose can be explained by looking at the implementation we obtain for the example platform given in Section 3. In the sequel, we call *tasks* (or *guest tasks*) the execution flows executing concurrently in the guest operating system (if any) and application layer, whatever the concurrency model of this system is.

The solution being technically involved, we first give an overview of the solution, ending in Section 4.3.2 with example execution paths for software and hardware requests. Then Sections 4.4 and 4.5 give more details on each part. Section 4.6 illustrates the solution with a complete example execution.

```
1 turn_adc_on ()        /* Provided by the adaptation layer to the upper layers. */
2   R = on_sw (adc_on);         /* Submit request 'adc_on' to the control layer. */
3   if (ack_a ∈ R) return success;
4   return error;           /* Return an error code if request has been refused. */
5
6 turn_adc_on ()                    /* (Ibid), however blocking until ADC is on. */
7   R = on_sw (adc_on);         /* Submit request 'adc_on' to the control layer. */
8   if (ack_a ∈ R) return success;
9   timer_wait (some time);       /* If request refused, block for some time... */
10  turn_adc_on ();                          /* ... and then retry.              */
```

Fig. 7. Two versions of a function of an ADC driver, which is to be included in the adaptation layer. adc_on is an input of the control layer that can be refused, in which case ack_a does not belong to the result of on_sw().

4.1. Main Structure

Figure 6 describes the main structure. From bottom to top, it shows: the hardware, the *Control Layer*, and the guest code (the guest OS, plus the application code). In order to communicate with the hardware, the guest uses dedicated *function calls* instead of direct low-level register operations. The element called *adaptation layer* on Figure 6 represents this modification. Modulo this slight modification of the hardware accesses, any existing OS can be ported on top of the control layer; Section 5 gives more details on this point.

The control layer implements the global control objectives for resource and power management by intercepting hardware requests (i.e., interrupt requests) and software requests (from the guest, via the adaptation layer). It maintains an up-to-date view of the current states of all the hardware devices. It presents a simplified view of the real hardware to the guest, by exporting a set of functions that may be called by the guest through the adaptation layer. These functions play the same role as the *hypercalls* of the paravirtualization approaches [Whitaker et al. 2002].

4.2. The Adaptation Layer

The adaptation layer is the part of the guest that needs to be modified in order to use the control layer. It comprises: (i) functions provided to the upper layers that issue software requests to the control layer (see turn_adc_on() example below); these functions can also register *callbacks* to be executed upon emission of a given event by the control layer; (ii) a run_guest() function exported to the control layer, whose purpose is to schedule and execute all runnable guest tasks, if any; this function returns when all tasks are blocked and all needed computations have been performed by the guest, meaning that the control layer can put the MCU in low-power mode (see Figure 10 below for details about the usage of this function). Figure 7 illustrates functions to be exported by the adaptation layer, for an ADC driver. on_sw() is a function (hypercall) provided by the control layer (see below).

4.3. Overview of the Control Layer

4.3.1. Structure. As depicted in Figure 8, the control layer is made of two parts: the *reactive part*, and the *event management* part.

The *reactive part* (also referred to as tick in the sequel) comes from all the automata of the drivers to be controlled, plus a controller. These drivers consist of BMAs as depicted in Section 2, which produce outputs triggering the execution of low-level code accessing registers of the devices. They also produce *output events* to provide information about the state of the devices to the adaptation layer. Inputs of these BMAs are twofold: requests emitted by the adaptation layer that trigger operations on the devices, plus *approval* signals from the controller, which restrict this behavior (details on approval signals and the controller are given in Section 4.5).

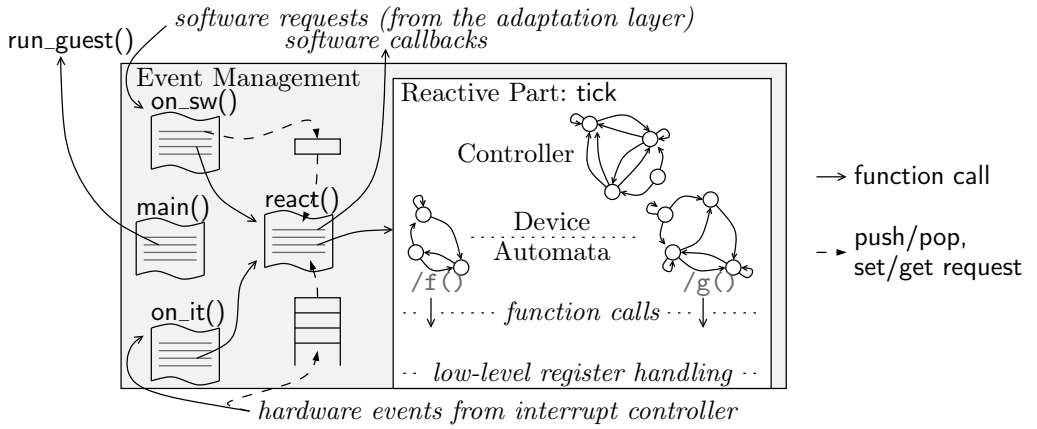

Fig. 8. Details of the Control Layer.

The tick is an object providing the methods init() and run_step(). It is the reactive kernel obtained from the compilation of the controller and the device automata, when composed in parallel (as mentioned in Section 2). The tick is entirely passive: it has to be called from the other part of the control layer; when called (using its run_step() method), it executes exactly *one* transition of the compiled automata, thus possibly executing some low-level code, and producing *output events*.

The *event management* part is in charge of managing a *queue* for hardware requests, handling the software ones, and building *input events* in order to call the tick. It also interprets the *output events* produced by the reactive part in order to send information to the upper layers. The *event management* part is made of the hardware request queue, plus several pieces of code. The hardware event queue is filled by the hardware only; the software request comes from the guest layer only. We first describe each piece of software. The complete behavior that results from their organization is best understood by looking at the two possible execution paths of Figure 9. The pieces of software are as follows:

— on_it() is the interrupt handler: its execution is launched by the occurrence of some interrupt (which has also posted an element in the hardware queue); it calls react();

— on_sw() is executed by the adaptation layer so as to emit a software request to be given to the tick; it mainly calls react() and returns some feedback about the request;

— react() consumes the elements in the queue of hardware requests and gather the pending software request, if any, in order to build an *input event* to be given to tick. It is a loop, calling tick.run_step() until the hardware queue becomes empty; when a software request s is given by on_sw(), and used as part of an event to run one tick, react() is able to interpret the *outputs* of the tick execution, and to transmit information to the adaptation layer both by executing callbacks and specifying the value returned by on_sw(s).

4.3.2. Example Execution Paths. Figure 9 describes the execution paths corresponding to software and hardware requests alone. Figure 18 and Section 4.6 illustrate a more general execution, where hardware and software requests may happen concurrently.

Example 1: Software Request Handling. Let us look at Figure 9-(a) first, and suppose that no hardware interrupt occurs. The guest code needs to perform an operation on the hardware (e.g., turn the ADC on); to do so, it calls a function of the adaptation layer, say, turn_adc_on() (see Figure 7). This function posts a software request *adc_on*, by calling the on_sw() function. Then, on_sw() immediately calls react(), that picks the new software request, and calls tick.run_step() with an event of the form: $adc\_on \;.\; \overline{x} \;.\; \overline{y} \ldots$ where $x, y, \ldots$

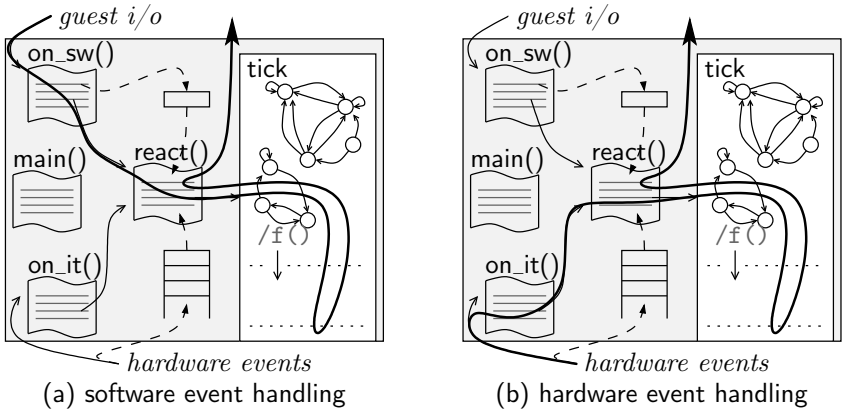

Fig. 9. Illustration of two different execution paths in the Control Layer. (a) Upon emission of a request by the adaptation layer by using on_sw(), the react() function is called, which triggers in turn an execution of tick.run_step(), and possibly some low-level code (e.g., f()). When tick.run_step() returns, some callbacks previously registered by the guest layer, may be executed. (b) Similarly, the notification of a hardware event (interrupt request) may trigger executions of the tick, and then some callbacks, through the on_it() function.

are the other possible software and hardware inputs (i.e., not input to the current reaction). The execution of the tick with such an event executes the appropriate transition from its current state, may execute some low-level code, and returns by providing the *output events* of the transition. react() analyzes this output, and may call some *callback* functions of the adaptation layer, to report about what happened when the software event was treated. In this execution path, the flow of control is not stopped during the treatment of one software request; as a result, the loop in react() iterates exactly once.

Example 2: Hardware Request Handling. Let us look at Figure 9-(b). A hardware interrupt occurs, i.e., the hardware puts an event irq_a in the hardware queue, and then "calls" on_it(). This may happen at any time, in particular while the processor is busy executing the software, including one call of react(). If a call to react() is executing currently, on_it() leaves the interrupt in the hardware queue, but does nothing to treat it (see below). If no call to react() is executing currently, then the software is preempted, and on_it() calls react(). That consumes the hardware event just posted in the hardware queue, and calls tick.run_step() with an event of the form: $irq_a \,.\, \overline{x} \,.\, \overline{y} \ldots$ where $x, y, \ldots$ are the software inputs, and the other hardware inputs. The execution of the tick with such an event executes the appropriate transition from its current state, may execute some low-level code, and returns by providing the *output events* of the transition. react() analyzes this output, and may call some functions of the adaptation layer. For instance, in case of an interrupt from the timer, the adaptation layer may have to wake up some task, or execute a callback, in the guest code.

4.4. Details on the Event-Management Part

Let us describe the left part of Figure 8, whose goal is to efficiently interleave the executions of the reactive part with those of the guest layer. It also manages all input requests, translating them into input events to be given to the reactive part, and conveys output information to the adaptation layer. The *events* that trigger executions of the reactive part are: *hardware events* (or *interrupts*), and *software requests*.

4.4.1. MCU State Management.

Running the Guest Layer. Figure 10 presents the main() function, which is always executed upon (re-)start of the MCU. Its role is to ensure that the MCU is in a low-power

```
1 main ()     /* Usual software entry point, executed with interrupts disabled. */
2   tick.init ();                          /* Initialize the reactive part.    */
3   while (true)
4     enable_interrupts ();          /* Enable interrupts before entering guest. */
5     run_guest (); /* Run guest until it requests entering low-power mode (LPM)
6                      by returning from this function.                         */
7     disable_interrupts (); /* Disable interrupts, because 'enter_lpm()' may
8                               need to perform some computation.               */
9     /* Here, we are guaranteed that no emission of software request is
10       possible until occurrence of next interrupt request.                   */
11    enter_lpm (); /* Enter LPM, and enable interrupts so that the MCU can be
12                     awakened: this function will return upon the next hardware
13                     event occurrence (cf. function 'on_it()' of Figure 12);
14                     also disables interrupts on return.                     */
```

Fig. 10. The main() function. Details for ensuring that we do not execute enter_lpm() if an interrupt occurs during execution of instructions between lines 5 and 7 have been omitted.

```
1 example_timer_irq_handler () /* Note that interrupts are automatically disabled
2                                 by hardware when this handler is executed.    */
3   acknowledge_irq ();                    /* Clear pending IRQ flag.          */
4   hw_queue.push (irq_expired);   /* Push hardware event corresponding to the
5                                 current interrupt request.                    */
6   on_it ();                              /* Next, call 'on_it()'.            */
```

Fig. 11. Example handler for a timer expiration interrupt. Notice details about interruption nesting have been omitted; yet such a behavior is still compatible with a use of the control layer.

```
1 on_it () /* Note that the event has already been pushed in 'hw_queue', and
2             interrupts are disabled.                                          */
3   if (! already_in_reaction) {
4     react ();                            /* Trigger reaction.               */
5     stay_awake (); /* Ensure the MCU will stay in active mode upon return    */
6   }               /* from interrupt, if it was in low-power mode hitherto.   */
```

Fig. 12. The on_it() function. It is called upon insertion of a new hardware event into the associated queue.

mode as often as possible. It is an endless loop, continuously executing the guest task(s) through the run_guest() function provided by the adaptation layer (typically, a call to the guest tasks manager, meaning that it runs until all tasks are blocked — see Section 4.2). If the latter function returns, then no more guest task is runnable, i.e., there is no more computation to perform. Until then, executions of both on_sw() and on_it() are possible.

Low-Power Mode Management. When the guest layer does not need to compute, the main() tries to enter the MCU in a low-power mode by using enter_lpm() (line 11 in Figure 10). Upon wake up of the MCU by one or more interrupts, associated interrupt handlers start executing.

Figure 11 depicts an example low-level interrupt handler in the control layer. One of its roles is to push the new request into the hardware requests queue, then call on_it() (lines 4 and 6). If the latter has triggered one or more reactions (i.e., some hardware requests have been emitted and taken into account), then it instructs the MCU to stay active upon return from the interrupt handler; the MCU goes back into its previous operating mode otherwise.

Going back to the code of the main() function in Figure 10, enter_lpm() returns if the MCU stays active after having returned from all pending interrupt handlers. Thus, guest tasks, which could have been released during executions of callbacks through on_it(), are executed across the subsequent call to run_guest().

```
1 on_sw (software_input)
2   sw_req.create (software_input);          /* Create a new software request.   */
3   disable_interrupts (); /* Disable interrupts to protect request management. */
4   sw_cell.set (sw_req);                  /* Assign the software request cell. */
5   react ();                               /* Trigger reaction.                 */
6   enable_interrupts ();                   /* Re-enable interrupts.             */
7   return sw_req.result (); /* Return the result that has been recorded in the
8                               software request object by 'react()'.           */
```

Fig. 13. The on_sw() function. The sw_req object records both the emitted input signal and the outputs resulting from the corresponding execution of tick. The latter data is assigned in react() (see Figure 14 below). sw_cell is a reference either pointing to a pending software request (not yet handled by react()), or nil if none exists.

```
1 react ()
2   already_in_reaction = true; /* Start the reaction (needed in 'on_it()').    */
3   all_outputs = empty_set ();
4   do { /* Build an input event by extracting the software request (if any)
5           and all the hardware events from the queue:                          */
6     input_event = build_input_event (hw_queue, sw_cell);
7     enable_interrupts ();                  /* Enabling interrupts allows new  */
8     outputs = tick.run_step (input_event); /* hardware events to be pushed    */
9     disable_interrupts ();                 /* into 'hw_queue' during the tick.*/
10    all_outputs.merge (outputs);     /* Union (bit-array operation actually). */
11    /* If there was a software request, then setup its result:               */
12    if (sw_cell != nil) sw_cell.set_result (outputs);
13    sw_cell = nil;              /* There is no more pending software request. */
14  } while (! hw_queue.is_empty ());
15  /* Here, hardware queue is empty, and no software request is pending.       */
16  already_in_reaction = false;            /* Notify we leave the reaction.    */
17  /* Eventually, launch callbacks associated with all emitted outputs:       */
18  launch_callbacks (all_outputs);
```

Fig. 14. The react() function.

4.4.2. Handling Requests from the Hardware. As seen above, the low-level interrupt handlers call on_it() when a hardware request is pushed into the hardware requests queue. This function, depicted in Figure 12, calls the react() function if it was not already running when the interrupt occurred (with the help of the already_in_reaction flag). Notice that after having called react(), it configures the MCU to stay in active mode on return of the current interrupt handler, so that any guest task potentially released can be executed (as stated in Section 4.4.1 above).

4.4.3. Handling Requests from the Software. The role of the on_sw() function (cf. Figure 13) is to trigger reactions upon emission of a software request by the adaptation layer. It returns the feedback information attached to the request by the react() function.

4.4.4. Running the Reactive Part. The react() function in Figure 14 behaves as follows: when called, it sets the already_in_reaction flag so that upcoming hardware events do not trigger reactions themselves (cf. on_it() function in Figure 12). The latter situation may happen if an interrupt occurs during the execution of tick.run_step() (instructions between lines 7 and 9 in react()). In such a case, hardware requests will be taken into account (i.e., popped from hw_queue and given to the reactive part) when the current execution of tick.run_step() ends. The already_in_reaction flag is reset at the end of the reaction.

The loop from lines 4 to 14 extracts and treats all pending requests from the hardware event queue until it becomes empty. It also treats any pending software request, if there is one. Each turn, all requests are popped from the queue and cell, and an input event is built and given to the reactive part (on line 8).

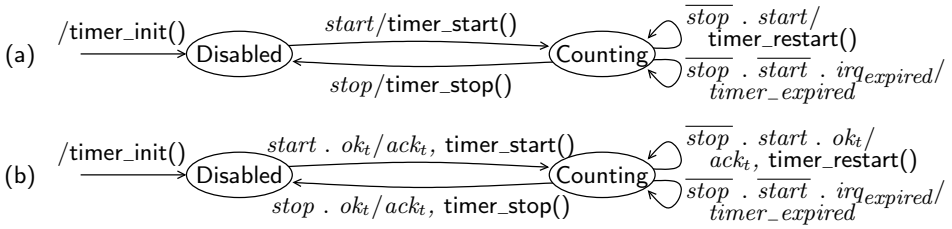

Fig. 15. Original (a) and controllable (b) device driver automata for a timer.

The output of this execution step of the tick is set as result of the software request that triggered it, if any (line 12).

Again, this result is merged with all outputs gathered during the preceding iteration of the loop (in set all_outputs, initialized to the empty set at the beginning of the function). This set serves on line 18 to launch all previously registered callbacks associated with the outputs that have been emitted during the reaction. Note that subsequent executions of react() are possible during the latter stage, either if interrupts occur (callbacks of the adaptation layer are executed with interrupts enabled), or if software requests are emitted.

4.5. Details of the Reactive Part

Let us now look at the details of the reactive part of Figure 8, sketched in Section 4.3.1.

4.5.1. Device Driver Automata. Designing the control path of a usual driver as an explicit automaton is quite natural. The states of this automaton reflect the operating modes of the device. The transitions are triggered by inputs that may be of two kinds: requests for changing modes, and hardware events. The idea is to guarantee that the state of the automaton always reflects the state of the device. The outputs on the transitions may represent low-level code (accesses to the device registers) that has to be executed to perform the real device operation.

In the sequel, we use the following syntax for a transition label: "$i_1 \,.\, \overline{i_2}$ / o_1, o_2, f()" where $i_1 \,.\, \overline{i_2}$ is an example Boolean formula built from the set of software and hardware inputs, o_1 and o_2 are example outputs, and f() is the call to some low-level code. For the sake of simplicity, self-loops with no outputs are not represented on the figures.

An important point here is the notion of *controllability*. Indeed, if we want to meet global control objectives, the requests from the software (and potentially some of the hardware events) should not always be accepted by the device driver. In the vocabulary of controller synthesis, it means that the automata to be controlled should have *controllable inputs*, otherwise the global control objective may be unfeasible. In the sequel, we design controllable automata for the devices. For one of the simplest (the timer), we explain how to add controllable inputs.

Figure 15-(a) is the automaton for a timer driver. timer_init(), timer_start(), timer_restart() and timer_stop() are low-level functions updating the timer operating mode and registers. The input $irq_{expired}$ is a hardware signal, whose meaning is the expiration of this timer. Finally, *stop* and *start* are input requests issued by the upper software layer to drive this device, and *timer_expired* is an output signal reflecting the expiration of the timer.

In Figure 15-(b), the automaton of Figure 15-(a) has been modified by introducing an approval input (ok_t): a controllable transition is triggered only when the input request holds and the controller emits ok_t.

In order to notify the requesting software, additional outputs are also used (ack_t in the timer example). They are emitted when controllable transitions are permitted.

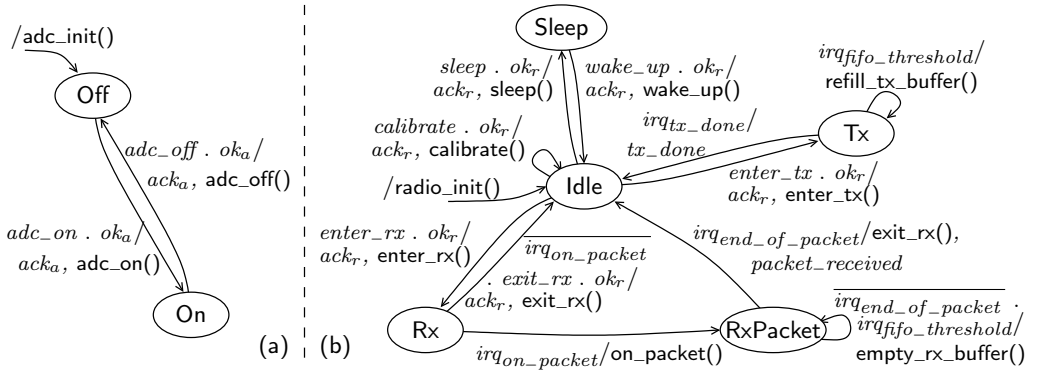

Fig. 16. Controllable ADC (a) and radio (b) automata. In the latter, ok_r and ack_r are approval and acknowledgment signals respectively. The driver is in Tx when the actual device transmits a packet, and in RxPacket when receiving one; it is in Rx when listening to the channel, and no packet was detected yet.

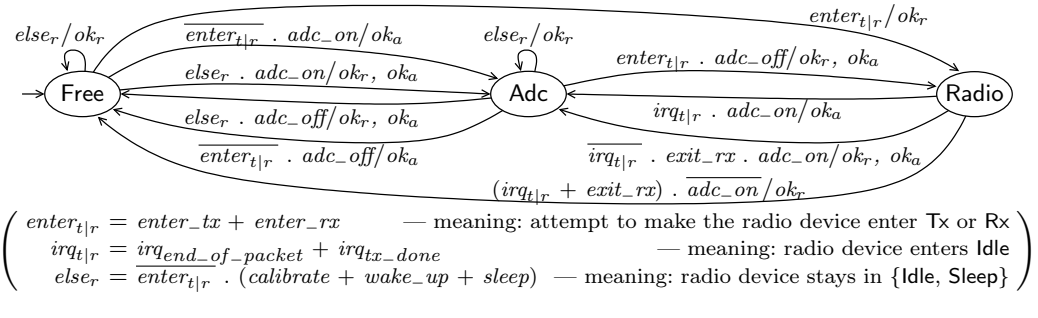

Fig. 17. Basic controller automaton for the drivers of Figure 16.

Note that, when in state Disabled and the condition $start \,.\, \overline{ok_t}$ occurs (i.e., the controller refuses the request *start*), there is a loop on state Disabled (not represented on Figure 15), which does not emit any output (in particular, ack_t).

Figure 16 describes a driver for an ADC, and a slightly simplified radio transceiver driver (without error handling transitions and wake-on radio feature) respectively.

4.5.2. The Controller Automaton. Figure 17 is an example of a controller designed from the radio transceiver and ADC automata of Figure 16. It ensures the exclusiveness between three energy-greedy states of the former (Tx, Rx and RxPacket) and the On mode of the latter. For instance, when the controller is in Radio (meaning that this device currently consumes energy), the ADC is necessarily in its Off state and can reach On (i.e., ok_a holds) iff the transceiver attains either Idle or Sleep.

4.5.3. The Compiled Automaton. Finally, the drivers and the controller are compiled into a single piece of code that forms the tick, and behaves as their product automaton. For the same example as before, the product automaton will be in state Free×Off×Idle when the controller, ADC and radio will be in Free, Off and Idle states respectively.

4.6. A Complete Example Execution

We illustrate on Figure 18 a complete interleaving of executions between the control layer and the guest layer for a simple example application, and a reactive part comprising device automata of Figures 15-(b), and 16, along with the controller of Figure 17. The time goes from left to right. For sake of simplicity, we trace the states of the controller, the ADC and

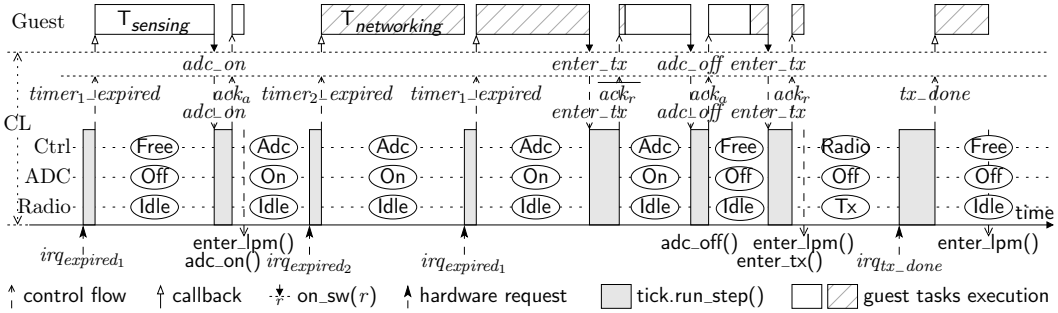

Fig. 18. Example execution. Inputs and outputs of the reactive part are shown only if they are relevant.

the radio transceiver between reactions. We also depict the role of the adaptation layer in this process by describing the software requests emitted to the control layer.

The guest layer is composed of two tasks; one performs the sensor management by using the ADC ($\mathsf{T}_{\mathit{sensing}}$), and the other periodically operates on the radio network ($\mathsf{T}_{\mathit{networking}}$). Executions of these two tasks are triggered by two distinct timers. Again, consider the CPU is initially in one of its low power modes.

First, the hardware event $irq_{expired_1}$ occurs, and triggers an execution of the reactive part. This outputs the $timer_1\_expired$ signal, and the react() function launches the associated callback. This callback restarts the $\mathsf{T}_{\mathit{sensing}}$ task execution, returns, and then the guest layer can continue its execution (the main() function calls run_guest()).

Upon emission of a software request by this task (a call to on_sw(adc_on)), a new reaction is triggered. Since the complete tick automaton is in state Free×Off×Idle at this time, this input triggers the transition taken as example in Section 4.5.3: the request is then approved and it turns the ADC on (the low-level adc_on() function is called). The request returns ack_a, and the task stops its execution: the main() function can then call enter_lpm().

Ibid, when $irq_{expired_2}$ occurs and wakes up the CPU, the task $\mathsf{T}_{\mathit{networking}}$ is scheduled. While computing, this task is interrupted by an $irq_{expired_1}$ hardware event, a reaction, and then a new call to the callback associated with $timer_1\_expired$. In turn, this callback enables a new execution of the sensing task. When exiting the interrupt handler, the networking task continues its execution, and tries to put the radio transceiver in its emitting mode (on_sw($enter\_tx$)). While the controller is in state Adc, it does not allow this software request, and its result is $\overline{ack_r}$ so the networking task can choose to yield and retry later.

Afterwards, the sensing task executes and turns the ADC off (on_sw(adc_off)); the associated reaction executes the adc_off() function, and updates the state of the controller. Eventually, the next emission of $enter\_tx$ is permitted and can succeed.

5. IMPLEMENTATION, CASE STUDY AND GUIDELINES FOR USING OUR SOLUTION

Let us now present a first assessment of our solution, based on a proof-of-concept implementation, and a case study to evaluate the impact of the adaptation of guest system software. We first describe some technical choices for the implementation. Next, we detail the case study, draw some guidelines, and measure the impact.

5.1. Technical Choices and Implementation

We have implemented the control layer on top of the hardware architecture described in Section 3. We have tested it using the cycle-accurate platform and network simulator provided with the Worldsens tools [Fraboulet et al. 2007].

5.1.1. Implementation of the Reactive Part. We have implemented a reasonable and usable set of device drivers so as to build up a working control layer on top of the Wsn430 hardware platform. The reactive part has been implemented using the LUSTRE academic compiler [Caspi et al. 1987]. We have manually designed a controller for all the devices and resources of the platform. It ensures simple safety properties like state exclusions for shared resource management, as well as reduction of current consumption peaks by avoiding reachability of global states when two or more peripherals (such as ADC and radio transceiver) are in their highest consumption modes.

5.1.2. Guest Layers. Regarding the guest layer, we have ported two operating systems that could also run on the bare hardware, onto our control layer implementation.

Targeting CONTIKI. We already depicted CONTIKI in Section 3. Its adaptation required writing a set of device drivers dedicated to the abstract hardware exposed by the control layer. The writing of these drivers is easy, whatever the strategy for handling rejected requests is: one could choose to retry requests after a given time, or return a dedicated error code to let the application processes select a more suitable strategy.

Targeting a Multithreading Operating System. The second operating system ported onto the control layer is a priority-based preemptive multithreading kernel we designed from scratch. Writing adapted device drivers in this guest was similar to writing those of CONTIKI, except for the task management and synchronization parts, as a result of the change of concurrency model.

5.2. Presentation of the Case Study

The goal is to adapt an existing implementation of a piece of software, so that it accesses the actual hardware through the control layer instead of the device drivers it is initially meant to use. We show in the following section that this process requires limited changes in the way the original software operates the hardware.

Original Software. In order to be convincing, we have chosen to port one of the most sophisticated system-level services in wireless sensor networks, that is, an implementation of a Medium Access Control (MAC) protocol, the part of the network stack responsible for the management of the physical layer (radio transceiver) in the OSI model. More particularly, it is an implementation of the X-MAC [Buettner et al. 2006], a low-power MAC protocol designed for WSNs. It comes from the SensTools suite [INRIA 2008] and has been designed to run without any particular operating system support (as described in Section 3). Also, it uses device drivers provided in the same suite as above, that drive the hardware platform described in Section 3. The original CC1100 transceiver driver is a rather basic one: it mostly acts as a wrapper of bus commands, and does not maintain the state of the device internally.

Required Radio Transceiver Driver Capabilities. In order to support all the possible operating mode transitions required by the X-MAC protocol, the radio device driver automaton integrated in the control layer needs to be slightly more complex than the one presented in Figure 16-(b) of Section 4.5.1. In practice, it has more controllable transitions. For instance, it comprises an additional state that represents the transmit operating mode from which the radio transceiver automatically jumps to listening mode when the packet is sent, without prior notification to the MCU. Additionally, it features a low-power listening (aka, "Wake on radio") state, which reflects a sleep state where the device is automatically awakened when it detects some radio signal.

5.3. X-MAC Original Implementation Details

We depict in Figure 19 the overall structure of the original X-MAC implementation.

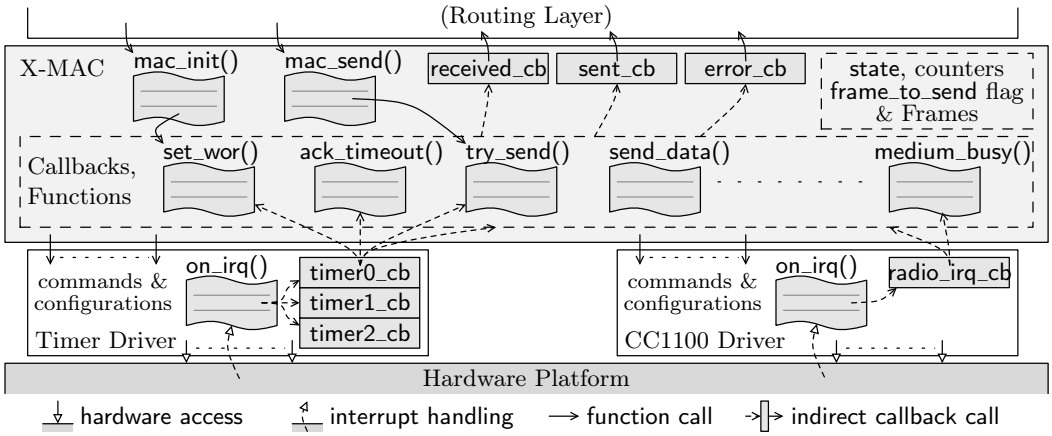

Fig. 19. Overall structure of the original X-MAC implementation, and the device drivers it uses. "Indirect callback calls" denote calls through function pointers that can be changed dynamically.

5.3.1. Programming Model and Implicit Requirements. After a first call to its initialization function mac_init(), all internal computations are performed either during interruption handling (i.e., a hardware event occurred) or on the initiative of the upper part of the network stack. In other words, it uses an *event-driven* programming model, where events are hardware interrupts, or direct function calls from other parts of the software stack.

The X-MAC can dynamically change the callbacks that the drivers call upon reception of an interrupt request. For instance, radio_irq_cb can be set to point to a function, which is then executed upon reception of the radio interrupt. At last, it also uses three alarm counters of the timer concurrently, each being able to trigger the execution of a different function, possibly periodically.

Using the X-MAC. Sending a packet requires calling mac_send(), which immediately returns a non-null integer in case of erroneous usage, or zero if the sending process has successfully started. Subsequently, the routing layer is asynchronously notified about the final status of this operation by executions of *callbacks*: the function pointed to by sent_cb is executed upon successful transmission of a frame; otherwise, the function pointed to by error_cb is called (e.g., the channel is too busy). Additionally, received_cb can also point to a function, which is then executed upon successful reception of a packet.

The return value of the callbacks can convey information on the subsequent state of the CPU as needed by the upper layer.

Internal Structure and Usage Constraints. This implementation is actually the encoding in an *event-driven* style of the X-MAC protocol state machine. Indeed, the state is encoded at any time by the mapping from the callback pointers provided by the device drivers to the callbacks of the X-MAC, plus a small amount of memory which persists across calls to X-MAC's functions. For instance, the set_wor() function, which puts the transceiver into a low-power listening mode of operation, also sets up read_frame() as the function to be called upon the next occurrence of the radio interrupt (radio_irq_cb — indicating that a packet has arrived when the device is low-power listening). Additionally, a state variable (encoding three very abstract states of the X-MAC — *sleeping*, *transmitting* or *receiving*), along with an additional frame_to_send flag and two counters, make up the remaining of the X-MAC internal state memory. Latter data serve two purposes: (i) detecting erroneous usage of the X-MAC; (ii) deciding what is the actual state of the protocol upon certain events.

Moreover, this implementation is not reentrant, basically meaning that any function or callback must never be called with interrupts enabled. It does not support concurrent re-

```
1 /* 'mac_send()' copies the buffer iff it succeeds...                    */
2 if (mac_send (buffer, buffer_length, destination_address) != 0)
3   { ... } /* Usage error (either the given packet is too long or there is one
4              to send already).                                           */
5 ... /* Do some more work, without detecting if the packet has effectively
6        been sent, or that an error occurred (both detectable by executions of
7        callbacks pointed to by 'sent_cb' or 'error_cb').               */
8 /* Potential failure: the MAC may still be sending the packet given on line 2.*/
9 if (mac_send (buffer, buffer_length, destination_address) != 0)
10  { ... }                                  /* Usage error again...      */
```

Fig. 20. Example erroneous usage of the X-MAC implementation: the call to mac_send() on line 9 will most probably fail, if the computation since line 2 is not long enough to let the MAC successfully send the first packet.

```
1 static uint16_t send_data(void) {      /* Sends a frame contained in txframe. */
2   timerB_unset_alarm(ALARM_PREAMBLE);     /* Unset alarm.                 */
3   cc1100_cmd_idle();                      /* Goto idle operating mode.    */
4   cc1100_cmd_flush_rx(); cc1100_cmd_flush_tx(); /* Flush FIFOs.           */
5   /* Dictate the device to enter idle mode upon end of transmission:      */
6   cc1100_cfg_txoff_mode(CC1100_TXOFF_MODE_IDLE);
7   cc1100_cmd_tx();                        /* Goto transmit mode           */
8   cc1100_fifo_put((uint8_t*)(&txframe.length), txframe.length+1);
9   /* Setup function to be executed upon end of transmission:              */
10  cc1100_gdo0_register_callback(send_done);
11  return 0;    /* Indicate we can stay in low-power mode, if we were already. */
12 }
```

Fig. 21. A piece of code of the original X-MAC implementation.

quests from the routing layer neither: as exemplified in Figure 20, any call to mac_send() must be followed by an execution of a function pointed to by either sent_cb or error_cb, before mac_send() can be called again.

5.3.2. Example Function. Figure 21 presents a listing of one of the functions of this MAC. It initiates the transmission of the contents of txframe, and is always explicitly called by other functions (i.e., never assigned as hardware event handler).

First, it disables a timer that can still be enabled at this execution point (line 2) and prepares the physical device for the transmission (lines 3 to 8). The behavior of the radio device is configured on line 6 to automatically go into idle mode once the packet is transmitted. The values of state and frame_to_send are left unchanged (a simple analysis of the call-graph indicates that their value always indicates that the MAC is currently transmitting a packet). Lastly, the callback associated with the radio interrupt (at this execution point always configured to be received upon end of transmission), is modified to the send_done() function on line 10.

5.4. Principles of the Adaptation

Identifying States of the Device. In order to adapt the X-MAC implementation we have described above, hardware devices operating mode transitions and configuration accesses need to be identified, and translated into appropriate software requests emitted to the control layer. This task is performed by reverse engineering the original implementation, and guessing the actual state of the device from the commands issued to its driver; since the latter does not track any state-related data, the required information needs to be incurred from the implicit control flow of the X-MAC (i.e., the call-graph is not sufficient due to the event-driven style of the encoding).

Once this information is extracted, the original commands are replaced with calls to functions of the adaptation layer, that issue state transition requests.

Refused Request Handling. As stated in Section 1.3.2, special care needs to be taken to handle potentially refused requests. Several choices are most often possible when adapting software. For instance, a (possibly unbounded) number of retries can be chosen when adapting a function that is never expected to fail (i.e., no error handling mechanism is provided to notify the upper software layers about failures). Doing so, the caller of such a procedure does not need to be modified.

5.5. Porting the X-MAC

Figure 22 presents the code resulting from an adaptation of the original code of Figure 21. Clearly, some of the commands sent to the radio transceiver can be directly translated into appropriate software requests sent to the control layer, as it is the case for the line 3 in the listing of Figure 21, resulting in the lines 22 to 26 in Figure 22. Further error handling mechanism could be easily employed since the X-MAC already exposed the error_cb callback. By using this solution, a request refused every try (in the radio_idle() or radio_send() wrappers) would be seen by the routing layer as an unsuccessful sending. Detailed information could also be given as argument to this callback, in order to indicate what caused the error, so the routing layer can take more suitable decisions.

Also, the command on line 7 in the original code of Figure 21 has led to the use of a transmit state that directly enters into *idle* mode upon end of emission. Hence, for our example radio transmitter driver, the radio_send() function called on line 29 in Figure 22 emits signal *enter_tx* to the control layer; this function is similar to radio_idle(), yet it also manages parameters of the request.

At last, dynamic interrupt handler assignments were straightforwardly translated into assignment of callbacks to outputs of the control layer. In Figure 22, line 28 registers send_done(); after executing this instruction, the given function will be called upon emission of *tx_done* by the control layer, i.e., upon end of transmission.

Debugging the X-MAC. Figure 23 presents an additional possible adaptation of the original code of Figure 21, that can be very helpful for debugging the X-MAC protocol implementation. In a context of non-controllable radio device driver in the control layer, this can reveal incorrect use of the radio, such as use of non-existent operating mode transitions.

5.6. Generalization

Adapting the X-MAC required converting 28 commands into appropriate calls to 9 distinct functions of the adaptation layer, each similar to the one presented in Figure 22. More than 85% of the original 700 lines of C code remained unchanged.

The X-MAC is a rather sophisticated piece of software, yet it required a limited amount of modifications. Moreover, the adaptation relies on general principles for the identification of *states*. We think we have now a reasonable guarantee that the method is widely applicable.

6. EVALUATION, DISCUSSION AND EXTENSIONS

The technical elements we have described in Section 5.1 constitute a complete proof of concept, for the implementation of centralized resource control policies in a paravirtualization framework. The implementation runs on top of a quite detailed emulator, which is a reasonable guarantee that it will also work on the real hardware.

Providing global property enforcement necessarily introduces some computation and memory overhead. In order to show that the proposed solution is practicable, we now estimate this overhead compared to available data about existing OSes for WSNs.

6.1. Quantitative Evaluation of our Solution

The memory footprint of the tick is about 1.5 to 2 KB (recall that the size of the tick is linear in the size of the individual automata descriptions, as stated in Section 2). Stack

```
/* Example wrapper function for radio management through the control layer.
   This function is part of the adaptation layer.                          */
extern status_t radio_idle (void) {
  /* Some arbitrary value defining the maximum number of retries; we could
     also make it an argument, retry infinitely, etc.:                     */
  const uint8_t max_tries = 8; uint8_t tries = 0;
  cl_outputs_v received;              /* This will be assigned by 'on_sw()'. */
  on_sw (cc1100_idle, & received);       /* First try.                      */
  while (! cl_outputs_test (received, cc1100_ack) && /* While we...        */
          tries++ != max_tries)      /* ... do not receive the acknowledgment: */
    ... /* Some code to be executed on forbidden request... note that it can
           be a call to a blocking function (e.g., for waiting some time);  */
    on_sw (cc1100_idle, & received);     /* Next try.                      */
  /* Return an error code on constant refusal:                             */
  return cl_outputs_test (received, cc1100_ack) ? SUCCESS : -EFAIL;
}

/* Adapted function for sending data, with advanced refusal handling.      */
static uint16_t send_data (void) {
  timerB_unset_alarm (ALARM_PREAMBLE);    /* Unset alarm.                  */
  /* Goto idle operating mode (this also flushes the buffers):             */
  if (radio_idle () != SUCCESS)
    /* The request has been refused, we can choose to notify the error;
       one could also try to enter in low-power listening here.       (*) */
    if (error_cb)                       /* Notify, if we have an error handler. */
      return error_cb ();         /* We could also pass a detailed error code. */
  /* Setup function to be executed upon end of transmission:               */
  radio_register_transmission_done_cb (send_done);
  if (radio_send ((uint8_t*)(&txframe.length), txframe.length+1) != SUCCESS)
    if (error_cb) return error_cb ();     /* Ibid (*)                      */
  return 0;             /* This value is ignored by the control layer actually. */
}
```

Fig. 22. Possible adaptation of original code of Figure 21, with advanced refused request handling. radio_idle() and radio_send() are parts of the adaptation layer: they emit input requests to the control layer, manage the data-path (e.g., payload), and deal with the outputs (by using, e.g., cl_outputs_test() that tests if a given signal belongs to a set of outputs from the control layer). radio_idle() is given as example: it tries to emit the *cc1100_idle* signal several times until it receives the acknowledgment, otherwise it returns an error code.

```
/* Adapted function for sending data (debug version).                      */
static uint16_t send_data (void) {
  timerB_unset_alarm (ALARM_PREAMBLE);    /* Unset alarm.                  */
  /* Goto idle operating mode (this also flushes the buffers):             */
  assert (radio_idle () == SUCCESS);
  /* Setup function to be executed upon end of transmission:               */
  radio_register_transmission_done_cb (send_done);
  assert (radio_send ((uint8_t*)(&txframe.length),txframe.length+1) == SUCCESS);
  return 0;             /* This value is ignored by the control layer actually. */
}
```

Fig. 23. Another way of adapting the original code of Figure 21, useful for debugging the usage of the radio device by the X-MAC. The assert() function is a debugging tool that usually prints an appropriate error message, and then blocks execution forever, if its argument evaluates to false.

space required for its computation is about 100 bytes, but it could be shared among all guest threads since there is always at most one call to the tick at a time. Other parts of the control layer mainly comprise the low-level code of the device drivers that would be in the guest otherwise. Rough implementation of the event-management part occupies 1 KB.

Let us now define the time overhead. The total time it takes to execute the tick once corresponds to: (i) executing code in the event-management part that feeds the tick; (ii) computing the next states of the compiled automata; (iii) possibly executing some low-level

Table I. Typical memory footprint of some existing OSes deployed on actual WSNs, with various sensor drivers and network modules. The "MtK/Control Layer" column represents our multithreading kernel and control layer implementation (TinyOS and RETOS data are taken from Cha et al. [2007]).

	TinyOS-1.1 (minimal kernel)	TinyOS-1.1	RETOS kernel	MtK/Control Layer
ROM	11.2 KB	21 KB	23.1 KB	24 KB
RAM	311 B	798 B	824 B	806 B

Table II. Typical overhead of ICEM decentralized shared resource arbiters and device power managers (more details about ICEM in Section 7.2), compared to our global control implementation.

ICEM with n arbiters & m power managers	one transition of the compiled automata in the tick
$\approx 350n + 400m$ CPU cycles	$\approx 1,600$ CPU cycles

code. (i) is negligible compared to (ii) since sets of requests are implemented as small bit-arrays; (iii) must be executed whatever the solution for resource control is. Hence the time overhead is the time needed for (ii). For instance, if the automaton of Figure 16-(b) is in state Idle, the overhead of on_sw(*enter_rx*) is the computation of the transitions of the compiled automata, without the time spent in low-level code like enter_rx().

More precisely, the time overhead is due to a sequence of Boolean operations (as in the body of run_step() in Figure 3): it is independent of the inputs, so the worst case can be obtained with one measure. We get it as the time of a call to tick.run_step(), with an event made of a software request that is refused, so that no low-level code is executed: it takes roughly 1,600 CPU cycles (200μs on an MSP430 clocked at 8MHz) in our proof-of-concept implementation.

Comparison with Existing Solutions. There are no available figures about other solutions for global control in WSN OSes, so we compare the results for our implementation with existing solutions involving localized resource control. We sum up in Table I the memory footprint of some of these OSes. Compared to these results, our solution involves a code size increase of less than 10%. This memory overhead is very realistic w.r.t. the benefits of global resource control that our approach can manage.

Table II compares overheads of decentralized control in the ICEM framework [Klues et al. 2007] and our proposal. The order of magnitude of the overhead introduced by our solution for global control and power management is reasonable, even if the tick is run for each software or hardware request.

6.2. Qualitative Evaluation

Developing a device driver for the control layer having the associated automaton in mind, reveals simpler than for classical OSes. Because in our approach, a clear distinction is made between low-level code that affects device operating modes, and effective application code.

The example adaptation process depicted in Section 5 shows that our solution involves a slight modification of parts of the guest software for refused request handling. Yet, we claim that the needed modifications have the advantage of making such error case handling explicit (and almost mandatory). It thus helps the software programmer in achieving a neat design, even for device drivers and OS services.

6.3. Towards Using Automatic Controller Synthesis

For this paper, we built controllers by hand, for simple safety properties like mutual exclusion. We used model-checking to check that the controller indeed enforces the properties. Yet, for more complex properties, and bigger systems, it is quite natural to try and use tools able to compute controllers automatically. There are not so many tools available, however, since research on controller synthesis is quite theoretical. In the synchronous community, we know of only one, called Sigali [Marchand et al. 2000], developed for the language Signal. Delaval et al. [2010] integrated this particular tool-chain into BZR, a language and

framework designed for investigating modular compilation. BZR has been used to generate system code in [Bouhadiba et al. 2011].

The controller synthesis *algorithms* are meant to produce the *most permissive controller*, which is often non-deterministic (for instance, to guarantee mutual exclusion, the controller makes a non-deterministic choice among potential users of the resource, when they are simultaneous; the simplest way to make the controller deterministic is to keep only one choice among the ones allowed). A tool like SIGALI implements those algorithms, plus a *determinization* phase, in order to produce a piece of C code that can be used as the actual controller. BZR also produces the controller as a C function FC.

We experimented with BZR, using it mainly as a wrapper of SIGALI. We first compose all the device automata at the synchronous language level, then compile the result into a C function FD. We call FC from FD, which is a trick to compose the controller with the device automata. Non-determinism is removed by choosing static priorities between the users of a resource. For the example described in the paper, the experiment is not entirely satisfactory. FC is quite big, thus leading to a noticeable overcost in the size of the code.

In fact, in our approach, we need to use the controller synthesis tool as a building block in a compiler, which is a non-intended use. In particular, we need to further compose the controller with other synchronous subprograms, and is not possible in the general case if it is given as an already sequentialized piece of C code. The ideal tool chain would produce the controller as a deterministic Mealy automaton with *oracles* (i.e., a non-deterministic choice between two transitions t_1 and t_2 is encoded by adding conditions i on t_1 and $\neg i$ on t_2, i being an additional *oracle input*). This automaton could then be composed freely with the driver automata. When a non-deterministic choice is due to two users asking for a resource at the same time, the choice could be resolved by composing the controller with a kind of fair *scheduler*, giving the resource to each user, in turn (technically, such a scheduler is an automaton emitting the oracle inputs of the controller). At the end, the synchronous program resulting from the composition of all these automata would be compiled into C.

The research groups working on SIGALI and BZR have similar concerns. They are currently working on more modular implementations of the controller synthesis algorithms, able to produce a controller as a program in some synchronous language, not as a C function. This is very promising for our purposes, because this will give us complete control on the removal of non-determinism, and it is likely to produce smaller code.

6.4. Deadlock Problem Considerations

Consider the following case, in our solution: a guest task T_1 (resp. T_2) controls a resource R_1 (resp. R_2), in such a way that T_1 (resp. T_2) is in state t_1 (resp. t_2) if and only if the resource R_1 (resp. R_2) is in state r_1 (resp. r_2). If a global property to be enforced forbids the state (r_1, r_2) for the resources, then the state (t_1, t_2) is forbidden for the guest tasks. If the application wants to reach the state (t_1, t_2), and depending on the way the global property is programmed, this may reveal as a deadlock, or as a livelock.

Anyway, enforcing global properties which are not compatible with the application, may lead to intrinsic deadlocks, whatever the solution used. In our solution, these deadlocks are easier to detect. Results from Wang et al. [2009] suggest that potential deadlocks can be detected by modeling the guest tasks into the control layer: a possible deadlock, as any unenforceable property, leads to unsuccessful synthesis of a controller. See next section for the problem of modeling tasks.

6.5. Extensions

The approach can be extended to treat more complex cases.

We may need to give some guest tasks a direct access to the hardware resources. Consider a multithreading OS guest, and a guest task that prints reports by directly sending characters through an UART, connected to a bus. Another device (e.g., the flash memory) uses the

same bus. We need some control for bus accesses, but since the task has a direct access to the hardware, it seems this cannot be done with our approach. In fact it can, by considering the task as an additional object similar to a device. We model its behavior with a two-state automaton (using the bus, or not) and controllable transitions, and we add this automaton to the set of automata for which we have to design a global controller. Then, we need to make sure that this two-state automaton always reflects the real state of the task. When the controller forbids a transition from the state "*not using the bus*" to the state "*using the bus*" of the task model, it also communicates with the guest scheduler, requesting him to remove the task from the list of eligible tasks. This mechanism is such that a global invariant is maintained: whenever the task is running (and accessing the bus), the control layer is in state "*using the bus*" and prevents other devices from using the bus.

Our approach can also be extended to ensure that the CPU is always in the lowest consumption mode possible. Note that the MCU should not be put in a mode M from which only external interrupts can exit, if there is no chance for such external interrupts to happen. Information on which external interrupts may occur is given by the states of the automata that model the hardware devices. We model the MCU by an automaton in which all the available mode changes are controllable transitions. The "*best low-power mode*" objective can then be stated as an invariant property (avoid the global states in which the MCU is in a mode M, and the interrupts that can occur in this state cannot exit M), plus a *quality* objective (among all the remaining states, choose the one corresponding to the lowest consumption mode).

7. RELATED WORK

7.1. Operating Systems for WSNs

Whereas originally designed for real-time embedded systems, MANTIS [Bhatti et al. 2005] has been ported to some WSN platforms. It is a priority-based preemptive multithreaded operating system and has shown the power of this concurrency model for WSNs by enabling the implementation of lengthy tasks. RETOS [Cha et al. 2007] and Nano-RK [Eswaran et al. 2005] are also multithreaded operating systems. However, neither MANTIS nor RETOS provide support for global resource management. Nano-RK provides static reservation mechanisms for energy and timing management of applications.

TinyOS [Hill et al. 2000] is the most widely known operating system for WSNs. It is fully component-based, event-driven and based on the nesC language [Gay et al. 2003], thus facilitating composition and reuse of previously written code. Component connections express the overall structure of the operating system along with the application. Nevertheless, building complex applications exploiting a broad range of the available devices supplied by a platform implies using dozens of components, hence leading to component bindings and interactions that are hard to apprehend globally.

7.2. Energy Management for WSN Nodes

Several works addressed the problem of energy management within a WSN node by means of a dedicated device driver architecture. The following ones are close to our proposal.

Klues et al. [2007] propose to address both concurrency and energy requirements in a single framework called ICEM, a core component of TinyOS 2.0. The key idea is to offer a driver interface based on (potentially concurrent) application I/O requests to better control the power states of the devices. From the application point of view, this concurrency level can be expressed by means of distinct driver classes. *Virtualized drivers* allow implicit concurrency between multiple users. Client requests are buffered and scheduled according to some desired properties (e.g., fairness), and a per-client state is maintained to control the power state of the device. *Shared drivers* also support multiple users, but they offer a lower level of interface in terms of *(power) locks*: each client should acquire a lock before using

a shared driver. A special component, the power manager, is responsible for implementing the energy management policy of a shared driver (e.g., powering off this driver as soon as its associated lock is idle). The ICEM's architecture leads to a decentralized energy and resource management scheme, split among driver classes.

Choi et al. [2008] suggested a global device driver architecture dedicated to multithreaded WSN OSes. This architecture also provides three kinds of driver models (offering several trade-offs between performance and complexity), and some global operating system services to control shared access and energy consumption through a device manager. This control is performed by means of a fixed set of dedicated request functions (either non blocking, or with a specified waiting time). Thanks to a centralized data structure indicating the current state of each device (and some specific device control functions) the device manager can assign the best suitable low-power state to the MCU and to each hardware element.

Although this proposal is close to our work in the sense that it offers a global control, it suffers from some drawbacks and limitations. The multithreaded device manager is certainly hard to write due to the concurrent shared access management and the use of multiple locks. Liveness problems may also occur, and not all devices are controlled (e.g., the timers are left uncontrolled, although they can impact the best low-power mode).

Other approaches have been proposed where the application logic is not involved in resource management decisions. ECOSystem [Zeng et al. 2002] is a general purpose operating system that integrates an explicit notion of "energy resource" into the scheduling mechanism of shared system devices. Eon [Sorber et al. 2007] can be viewed as an *energy-aware* data-flow programming language, where flow paths within programs are annotated with energy states by the programmer. Then, at runtime, Eon "adapts" the application code by selecting a suitable dataflow path according to current energy availability.

Pixie OS [Lorincz et al. 2008] is a more recent operating system dedicated to sensor nodes. It borrows some ideas from ECOSystem and Eon. Its purpose is to enable some *resource-aware programming model* with respect to energy, radio bandwidth, storage, etc. It relies on a dataflow model plus a notion of *resource tickets* to abstract the allocation of physical resources. Resource management policies can be enforced by means of (dedicated) resource brokers that deliver resource tickets to the application. The notion of "broker" is rather close to the global controller we propose. However, it differs in several points: first, brokers are dedicated to specific resources, meaning that a broker competition could be necessary to enforce global properties (related to several resources); second, correctly estimating an energy quantum for a given work unit could be a difficult task, and a too conservative approach could degrade the node performance; finally, there is no general technique for designing a "correct" broker with respect to a given policy.

7.3. Automated Control and Operating Systems

The research community on computing systems, in particular operating and distributed systems, has been showing interest for the use of control theory for some years now. Abdelzaher et al. [2008] present a very good introduction to the field, and exposes several applications, among which power control.

7.4. Formal Models for Driver Design

In [Wang et al. 2009], formal models of drivers are used as an abstract specification, from which the code can be produced. The whole approach is comparable to our use of automata labeled by function calls for the low-level programs.

In [Chis et al. 2009], formal models are used to *map* a MAC algorithm on top of a complex radio device; the radio device has more states than the functional states that matter for the software. The proposed method may allow, for instance, to specify in the MAC algorithm that the radio should go from idle to transmit mode; the formal model of the radio device is then used to transform this functional behavior into a more complex behavior of the radio,

which needs to go through various states of different energy levels between idle and transmit. The mapping algorithm could be formulated as a control objective, with controlled events forcing transitions instead of inhibiting them, but this would need further investigation.

8. CONCLUSIONS AND FURTHER WORK

We proposed a software architecture for embedded systems, allowing for global control of the hardware devices. The advantages of the approach are: (i) a clear expression of the global control objectives, that helps designing the global controller; (ii) the use of a synchronous language for the control layer, which makes it possible to compile the set of drivers and the controller into a piece of statically-scheduled efficient code; (iii) the easy extensibility of an existing control layer. Our method requires a slight adaptation of the application software, but does not change the way it is designed and programmed. We showed how to modify an existing piece of software to run it on top of our control layer; the method used is sufficiently general to be applicable to a large set of low-level software.

Further work will follow three main directions. First, we will study situations in which the applications may need to *book* some resources in advance, to prevent their requests from being canceled; this can be done with more complex automata for the resources, but essentially the same kind of controller as presented in this paper. Second, we will follow the advances in automatic controller synthesis tools, and adapt them to our context. A longer term perspective is to implement the whole software of a node in some synchronous language, and to perform static analysis and then static scheduling. When the application is written using event-driven programming, as in TinyOS for instance, it would not change the practice a lot to write it in the kind of automaton-based language we used here (the automata exchange signals in a synchronous way to express the synchronization, and this mechanism is indeed compiled; their transitions are also labeled by calls to pure C code, allowing for an easy reuse of, e.g., the protocol stack). This would provide both: structured event-driven programming with no additional runtime cost, and a formal model of the whole system that can be analyzed before it is deployed.

References

Abdelzaher, T., Diao, Y., Hellerstein, J. L., Lu, C., and Zhu, X. 2008. Introduction to control theory and its application to computing systems. In *Performance Modeling and Engineering*, Z. Liu and C. H. Xia, Eds. Springer US, Boston, MA, USA, 185–215.

Altisen, K., Clodic, A., Maraninchi, F., and Rutten, É. 2003. Using controller-synthesis techniques to build property-enforcing layers. In *Proceedings of the 12th European Conference on Programming*. ESOP'03. Springer-Verlag, Berlin, Heidelberg, 174–188.

Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Le Guernic, P., and de Simone, R. 2003. The synchronous languages 12 years later. *Proc. of the IEEE 91,* 1, 64–83.

Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., and Han, R. 2005. MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl. 10*, 563–579.

Bouhadiba, T., Sabah, Q., Delaval, G., and Rutten, É. 2011. Synchronous control of reconfiguration in fractal component-based systems — a case study. In *Proceedings of the 11st ACM International Conference on Embedded Software*. EMSOFT '11. ACM, New York, NY, USA.

Buettner, M., Yee, G. V., Anderson, E., and Han, R. 2006. X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. SenSys '06. ACM, New York, NY, USA, 307–320.

Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. 1987. LUSTRE: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. ACM, New York, NY, USA, 178–188. The Lustre compiler is available at `http://www-verimag.imag.fr/The-Lustre-Toolbox.html`.

Caspi, P., Scaife, N., Sofronis, C., and Tripakis, S. 2008. Semantics-preserving multitask implementation of synchronous programs. *ACM Trans. Embed. Comput. Syst. 7*, 15:1–15:40.

Cha, H., Choi, S., Jung, I., Kim, H., Shin, H., Yoo, J., and Yoon, C. 2007. RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In *Proceedings of the 6th*

International Conference on Information Processing in Sensor Networks. IPSN '07. ACM, New York, NY, USA, 148–157.

Chandra, V., Huang, Z., and Kumar, R. 2003. Automated control synthesis for an assembly line using discrete event system control theory. *IEEE Trans. Syst. Man and Cybernetics 33,* 2, 284–289.

Chis, A., Fleury, E., and Fraboulet, A. 2009. An optimized MAC layer to physical device mapping methodology. In *Proceedings of the 6th International Conference on Mobile Technology, Application & Systems*. Mobility '09. ACM, New York, NY, USA, 47:1–47:8.

Choi, H., Yoon, C., and Cha, H. 2008. Device driver abstraction for multithreaded sensor network operating systems. In *Proceedings of the 5th European Conference on Wireless Sensor Networks*. EWSN'08. Springer-Verlag, Berlin, Heidelberg, 354–368.

Delaval, G., Marchand, H., and Rutten, É. 2010. Contracts for modular discrete controller synthesis. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '10. ACM, New York, NY, USA, 57–66.

Dunkels, A., Gronvall, B., and Voigt, T. 2004. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. LCN '04. IEEE Computer Society, Washington, DC, USA, 455–462.

Dunkels, A., Schmidt, O., Voigt, T., and Ali, M. 2006. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. SenSys '06. ACM, New York, NY, USA, 29–42.

Eswaran, A., Rowe, A., and Rajkumar, R. 2005. Nano-RK: An energy-aware resource-centric RTOS for sensor networks. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*. IEEE Computer Society, Washington, DC, USA, 256–265.

Fraboulet, A., Chelius, G., and Fleury, E. 2007. Worldsens: development and prototyping tools for application specific wireless sensors networks. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*. IPSN '07. ACM, New York, NY, USA, 176–185.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. 2003. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI '03. ACM, New York, NY, USA, 1–11.

Girault, A. and Rutten, É. 2009. Automating the addition of fault tolerance with discrete controller synthesis. *Form. Methods Syst. Des. 35*, 190–225.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev. 34*, 93–104.

INRIA 2008. *SensTools*. INRIA. `http://senstools.gforge.inria.fr/`.

Klues, K., Handziski, V., Lu, C., Wolisz, A., Culler, D., Gay, D., and Levis, P. 2007. Integrating concurrency control and energy management in device drivers. *SIGOPS Oper. Syst. Rev. 41*, 251–264.

Lorincz, K., Chen, B.-r., Waterman, J., Werner-Allen, G., and Welsh, M. 2008. Resource aware programming in the Pixie OS. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*. SenSys '08. ACM, New York, NY, USA, 211–224.

Maraninchi, F. and Rémond, Y. 2001. Argos: an automaton-based synchronous language. *Comput. Lang. 27,* 1-3, 61–92.

Marchand, H., Bournai, P., Le Borgne, M., and Le Guernic, P. 2000. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic Systems 10*, 325–346.

Ramadge, P. J. G. and Wonham, W. M. 1989. The control of discrete event systems. *Proc. of the IEEE 77,* 1, 81–98.

Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M. D., and Berger, E. D. 2007. Eon: a language and runtime system for perpetual systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*. SenSys '07. ACM, New York, NY, USA, 161–174.

Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., and Mahlke, S. 2009. The theory of deadlock avoidance via discrete control. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '09. ACM, New York, NY, USA, 252–263.

Whitaker, A., Shaw, M., and Gribble, S. D. 2002. Scale and performance in the Denali isolation kernel. *SIGOPS Oper. Syst. Rev. 36*, 195–209.

Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. 2002. ECOSystem: managing energy as a first class operating system resource. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS-X. ACM, New York, NY, USA, 123–132.